Research Report

On Computing the Data Cube

Sunita Sarawagi *Rakesh Agrawal* *Ashish Gupta*

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).

Research Division
Yorktown Heights, New York • San Jose, California • Zurich, Switzerland

On Computing the Data Cube

Sunita Sarawagi[*] *Rakesh Agrawal* *Ashish Gupta*[†]

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

ABSTRACT: On-Line Analytical Processing (OLAP) applications often require computation of multiple related group-bys. This paper presents fast algorithms for computing a collection of group-bys. We focus first on a special case of the aggregation problem—computation of the cube operator. The cube operator requires computing group-bys on all possible combinations of a list of attributes. Our algorithms extend hash-based and sort-based grouping methods with several optimizations, like combining common operations across multiple group-bys, caching, and using pre-computed group-bys for computing other group-bys. Empirical evaluation using real-life OLAP data shows that the resulting algorithms yield a factor of two to eight improvement over straightforward methods and have running times very close to estimated lower bounds.

We then extend our algorithms to compute a subset of a cube in which the required set of aggregates does not include all combinations of attributes. Finally, we extend our algorithms to handle the common OLAP case in which attributes are grouped along hierarchies defined on them.

[*] Current Address: University of California, Berkeley, California.
[†] Current Address: Oracle Corporation, Redwood City, California.

1. Introduction

Aggregation is a predominant operation in decision support database systems. On-Line Analytical Processing (OLAP) databases [CCS93] often need to summarize data at various levels of detail and on various combinations of attributes. Recently, [GBLP95] introduced the *data cube* operator for conveniently supporting such aggregates in OLAP databases. The data cube operator computes group-bys corresponding to all possible combinations of a list of attributes. For instance, consider a common OLAP table `Transactions` with four attributes `product(P)`, `date(D)`, `market(M)` and `sales(S)` that records the sales value of a company's products on various dates for various markets. We are interested in finding the sum of sales for each product, for each market, for each date, for each product-market combination, for each market-date combination, and so on. This collection of aggregate queries can be conveniently expressed using the cube-operator as follows:

```
select sum(S) from Transactions cube-by P, D, M
```

This query will result in the computation of $2^3 = 8$ group-bys: `PDM`, `PD`, `PM`, `DM`, `D`, `M`, `P` and `all`, where `all` denotes the empty group-by. The straightforward way to support the above query is to rewrite it as a collection of eight group-by queries and execute them separately. There are several ways in which this simple solution can be improved.

In this paper, we present fast algorithms for computing multiple aggregates. Our main focus is on computation of the cube operator but many of our conclusions hold for the general case of the computation of aggregates of a subset of all possible combinations of attributes. We extend our cube-operator algorithms for this general case. In addition, in typical OLAP applications, attributes have hierarchies defined on them [CCS93]. Thus, `product` $\rightarrow$ `type` $\rightarrow$ `category` is a hierarchy on `product` that specifies various aggregation levels for products. For instance, "ivory" and "irish spring" both are of type "soap." Furthermore, "soap" and "shampoo" both are in category "personal hygiene" products. We also extend our cube-operator algorithms for aggregations along hierarchies.

For this paper, we assume that the aggregating functions are *distributive* [GBLP95], that is, they allow the input set to be partitioned into disjoint sets that can be aggregated separately and later combined. Examples of distributive functions include `max`, `min`, `count`, and `sum`. The proposed algorithms are also applicable to the *algebraic* aggregate functions [GBLP95], such as `average`, that can be expressed in terms of other distributive functions (`sum` and `count` in the case of `average`). As pointed out in [GBLP95], other aggregate functions (*holistic* functions of [GBLP95]) such as `median` cannot be computed in parts and combined.

There are two basic methods for computing a group-by: (1) the sort-based method and (2) the hash-based method [Gra93]. We will adapt these methods to compute multiple group-bys by incorporating the following optimizations:

1. **Smallest-parent:** This optimization, first proposed in [GBLP95], aims at computing a group-by from the smallest previously computed group-by. In general, each group-by can be computed from a number of other group-bys. Figure 1 shows a four attribute cube ($ABCD$) and the options for computing a group-by from a group-by having one more attribute called its *parent*. For instance, AB can be computed from ABC, ABD or $ABCD$. ABC or ABD are clearly better choices for computing AB. In addition, even between ABC and ABD, there can often be big difference in size making it critical to consider size in selecting a parent for computing AB.
2. **Cache-results:** This optimization aims at caching (in memory) the results of a group-by from which other group-bys are computed to reduce disk I/O. For instance, for the cube in Figure 1,

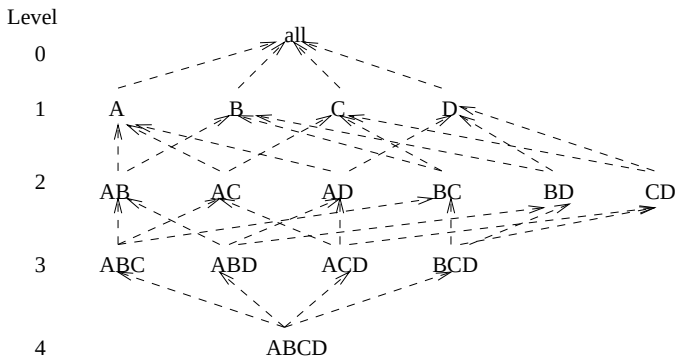


Figure 1: A search lattice for the cube operator

having computed ABC, we compute AB from it while ABC is still in memory.

3. **Amortize-scans:** This optimization aims at amortizing disk reads by computing as many group-bys as possible, together in memory. For instance, if the group-by $ABCD$ is stored on disk, we could reduce disk read costs if all of ABC, ACD, ABD and BCD were computed in one scan of $ABCD$.
4. **Share-sorts:** This optimization is specific to the sort-based algorithms and aims at sharing sorting cost across multiple group-bys.
5. **Share-partitions:** This optimization is specific to the hash-based algorithms. When the hash-table is too large to fit in memory, data is partitioned and aggregation is done for each partition that fits in memory. We can save on partitioning cost by sharing this cost across multiple group-bys.

For OLAP databases, the size of the data to be aggregated is usually much larger than the available main memory. Under such constraints, the above optimizations are often contradictory. For computing B, for instance, the first optimization will favor BC over AB if BC is smaller but the second optimization will favor AB if AB is in memory and BC is on disk.

Contributions.

The main contributions of this paper are:

- *Design of fast algorithms for computing the data cube.* We listed above five optimizations for combining multiple group-bys to efficiently compute a cube. We combine these (often contradictory) optimizations into a single algorithmic framework. The resulting algorithms exhibit significant performance improvement over straightforward algorithms that compute each group-by separately. Empirical evaluations show that they have running times very close to estimated lower bounds. Between the sort-based and hash-based algorithms, we determine that the sparsity of a cube is a crucial differentiator.
- *Extend the cube algorithms for computing a specified subset of the group-bys in a cube.* We identify a reduction of the problem to the *minimum steiner tree* [GJ] problem. This enables us to find plans that consider computation of intermediate group-bys that are not part of the specified subset but can lead to smaller total cost.
- *Handle aggregation hierarchies associated with attributes*. We give extensions to our algorithms in which attributes have hierarchies defined on them

Related Work.

Methods of computing *single* group-bys have been well studied (see [Gra93] for a survey), but little work has been done on optimizing a collection of related aggregates. [GBLP95] gives some rules of thumb to be used in an efficient implementation of the cube operator. These include the smallest parent optimization and partitioning of data by attribute values that we adopt in our algorithms. However, the primary focus in [GBLP95] is on defining the semantics of the cube operator [GBLP95]. There are reports of on-going research related to the data cube in directions complementary to ours: [HRU96] [GHRU96] presents algorithms for deciding what group-bys to pre-compute; [SR96] and [JS96] discuss methods for indexing pre-computed summaries to allow efficient querying. A concurrent work on computing data cube is reported in [DANR96].

Aggregate pre-computation is quite prevalent in statistical databases [Sho82]. Research in this area has considered various aspects of the problem starting from developing a model for aggregate computation [CM89], indexing pre-computed aggregates [STL89] and incrementally maintaining them [Mic92]. However, there is no published work, to the best of our knowledge, in the statistical database literature on methods for optimizing the computation of related aggregates.

Organization of the paper. In Section 2 we present the sort-based algorithm for computing the cube and in Section 3 we present the hash-based algorithm. In Section 4 we present experimental evaluation of our algorithms and present a comparison between sort and hash-based algorithms. In Section 5 we suggest extensions to the cube algorithms. We make some concluding remarks in Section 6.

2. Sort-based methods

In this section, we present the sort-based algorithm that incorporates the optimizations listed in Section 1. We include the optimization **share-sort** by using data sorted in a particular order to compute all group-bys that are prefixes of that order. For instance, if we sort the raw data on attribute order $ABCD$, then we can compute group-bys $ABCD$, ABC, AB and A without additional sorts. However, this decision could conflict with the optimization **smallest-parent**. For instance, the smallest parent of AB might be BDA although by generating AB from ABC we are able to share the sorting cost. It is necessary, therefore, to do global planning to decide what group-by is computed from what and the attribute order in which it is computed. We propose an algorithm called *PipeSort* that combines the optimizations share-sorts and smallest-parent to get the minimum total cost.

The PipeSort algorithm also includes the optimizations **cache-results** and **amortize-scans** to reduce disk scan cost by executing multiple group-bys in a *pipelined* fashion. For instance, consider the previous example of using data sorted in the order $ABCD$ to compute prefixes $ABCD$, ABC, AB and A. Instead of computing each of these group-bys separately, we can compute them in a pipelined fashion as follows. Having sorted the raw data in the attribute order $ABCD$, we scan the sorted data to compute group-by $ABCD$. Every time a tuple of $ABCD$ is computed, it is propagated up the pipeline to compute ABC; every time a tuple of ABC is computed, it is propagated up to compute AB, and so on. Thus, each pipeline is a list of group-bys all of which are computed in a single scan of the sort input stream. During the course of execution of a pipeline we need to keep only one tuple per group-by in the pipeline in memory.

Algorithm PipeSort. Assume that for each group-by we have an estimate of the number of distinct values. A number of statistical procedures (e.g., [HNSS95]) can be used for this purpose. We discuss in the Appendix the estimation procedure we use in our implementation.

The input to the algorithm is the *search lattice* defined as follows.

Search Lattice. A search lattice for a data cube [HRU96] is a graph where a vertex represents a group-by of the cube. A directed edge connects group-by i to group-by j whenever j can be generated from i and j has exactly one attribute less than i (i is called the parent of j). Thus, the out-degree of any node with k attributes is k. Figure 1 is an example of a search lattice. Level k of the search lattice denotes all group-bys that contain exactly k attributes. The keyword `all` is used to denote the empty group-by (Level 0). Each edge in the search lattice e_{ij} is labeled with two costs. The first cost $S(e_{ij})$ is the cost of computing j from i when i is not already sorted. The second cost $A(e_{ij})$ is the cost of computing j from i when i is already sorted.

The output, O of the algorithm is a subgraph of the search lattice where each group-by is connected to a single parent group-by from which it will be computed and is associated with an attribute order in which it will be sorted. If the attribute order of a group-by j is a prefix of the order of its parent i, then j can be computed from i without sorting i and in O, edge e_{ij} is marked A and incurs cost $A(e_{ij})$. Otherwise, i has to be sorted to compute j and in O, e_{ij} is marked S and incurs cost S_{ij}. Clearly, for any output O, there can be at most one out-edge marked A from any group-by i, since there can be only one prefix of i in the adjacent level. However, there can be multiple out-edges marked S from i. The objective of the algorithm is to find an output O that has minimum sum of edge costs.

Algorithm. The algorithm proceeds level-by-level, starting from level $k = 0$ to level $k = N - 1$, where N is the total number of attributes. For each level k, it finds the best way of computing level k from level $k + 1$ by reducing the problem to a weighted bipartite matching problem[1] [PS82] as follows.

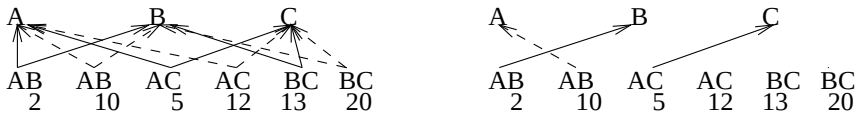
We first transform level $k + 1$ of the original search lattice by making k additional copies of each group-by in that level. Thus each level $k + 1$ group-by has $k + 1$ vertices which is the same as the number of children or out-edges of that group-by. Each replicated vertex is connected to the same set of vertices as the original vertex in the search lattice. The cost on an edge e_{ij} from the original vertex i to a level k vertex j is set to $A(e_{ij})$ whereas all replicated vertices of i have edge cost set to $S(e_{ij})$. We then find the minimum [2] cost matching in the bipartite graph induced by this transformed graph. In the matching so found, each vertex h in level k will be matched to some vertex g in level $k + 1$. If h is connected to g by an $A()$ edge, then h determines the attribute order in which g will be sorted during its computation. On the other hand, if h is connected by an $S()$ edge, g will be re-sorted for computing h.

For illustration, we show how level 1 group-bys are generated from level 2 group-bys for a three attribute search lattice. As shown in Figure 2(a), we first make one additional copy of each level 2 group-by. Solid edges represent the $A()$ edges whereas dashed edges indicate the $S()$ edges. The

[1] The weighted bipartite matching problems is defined as follows: We are given a graph with two disjoint sets of vertices V_1 and V_2 and a set of edges E that connect vertices in set V_1 to vertices in set V_2. Each edge is associated with a fixed weight. The weighted matching problem selects the maximum weight subset of edges from E such that in the selected subgraph each vertex in V_1 is connected to at most one vertex in V_2 and viceversa.

[2] Note we can covert a mininum weight matching to a maximum weight matching defined earlier by replacing each edge weight w by $max(w) - w$ where $max(w)$ is the maximum edge cost.

number underneath each vertex is the cost of all out-edges from this vertex. In the minimum cost matching (Figure 2(b)), A is connected to AB with an $S()$ edge and B by an $A()$ edge. Thus at level 2, group-by AB will be computed in the attribute order BA so that B is generated from it without sorting and A is generated by resorting BA. Similarly, since C is connected to AC by an $A()$ edge, AC will be generated in the attribute order CA. Since, BC is not matched to any level-1 group-by, BC can be computed in any order.



(a) Transformed search lattice (b) Minimum cost matching

Figure 2: Computing level 1 group-bys from level 2 group-bys in a 3 attribute cube

We use the algorithm in [PS82] for finding the minimum cost matching in a bipartite graph[3]. The complexity of this algorithm is $O(((k+1)M_{k+1})^3)$, where M_{k+1} is the number of group-bys in level $k+1$.

PipeSort:
(Input: search lattice with the A() and S() edges costs)
For level $k = 0$ to $N-1$ /* N is the total number of attributes */
    /* determine how to generate level k from level $k+1$ */
    Generate-Plan($k+1 \rightarrow k$);
    For each group-by g in level $k+1$
        Fix the sort order of g as the order of the level k group-by
            that is connected to g by an A() edge;

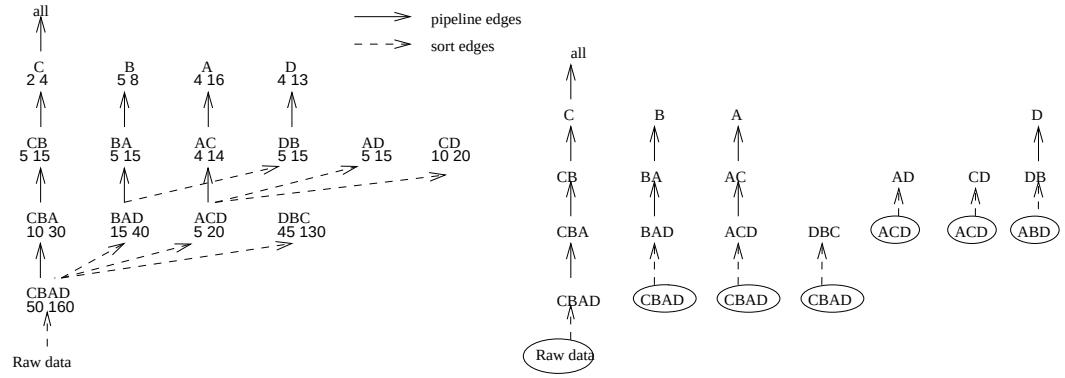
Generate-Plan($k+1 \rightarrow k$)
    Create k additional copies of each level $k+1$ vertex;
    Connect each copy vertex to the same set of level k vertices as the original vertex;
    Assign cost $A(e_{ij})$ to edge e_{ij} from the original vertex and cost $S(e_{ij})$ to edge from the copy vertex;
    Find the minimum cost matching on the transformed level $k+1$ with level k;

Example: We illustrate the PipeSort algorithm for the four attribute lattice of Figure 1. For simplicity, assume that for a given group-by g the costs A() and S() are the same for all group-bys computable from g. The pair of numbers underneath each group-by in Figure 3 denote the A() and S() costs. Solid edges denote A() edges and dashed edges denote S() edges. For these costs, the graph in Figure 3(a) shows the final minimum cost plan output by the PipeSort algorithm. Note that the plan in Figure 3(a) is optimal in terms of the *total cost* although the *total number of sorts* is suboptimal [4].

In Figure 3(b) we show the pipelines that are executed. Sorts are indicated by ellipses. We would first sort data in the order $CBAD$. In one scan of the sorted data, $CBAD$, CBA, CB, C and `all` would be computed in a pipelined fashion. Then group-by $ABCD$ would be sorted into the new order $BADC$ and thereafter BAD, BA and B would be computed in a pipelined fashion.

[3] The code for the matching algorithm is available from `ftp-request@theory.stanford.edu`

[4] The optimal number of sorts for a N attribute cube is $C(N, N/2)$ where $C(N,i) = \frac{N!}{i!(N-i)!}$ denotes the total number of group-bys containing i attributes [GGL95].



(a) The minimum cost sort plan

(b) The pipelines that are executed

Figure 3: Sort-based method for computing a four attribute cube

We can make the following claims about algorithm PipeSort.

Claim 2.1. *Generate-plan() finds the best plan to get level k from level $k+1$.*

PROOF. *Follows by construction assuming a cost function where the cost of sorting a group-by does not depend on the order in which the group-by is already sorted.*

Claim 2.2. *Generate-plan($k+1 \rightarrow k$) does not prevent Generate-plan($k+2 \rightarrow k+1$) from finding the best plan.*

PROOF. *After we have fixed the way to generate level k from level $k+1$ the only constraint we have on level $k+1$ is the order in which the group-bys should be generated. This ordering does not affect the minimum matching solution for generating level $k+1$ from $k+2$. After finding the best solution for generating level $k+1$ from level $k+2$, we can always change the order in which each group-by should be generated (as dictated by level k solution) without affecting the minimum cost.*

Note that PipeSort computes each group-by from a group-by occurring only in the immediately preceding level. Although the level-by-level approach is not provably optimal, we have not been able to find any case where generating a group-by from a group-by not in the preceding level leads to a better solution. Our experiments reported in Section 4 also show that our solution is very close to empirically estimated lower bounds for several datasets.

Further Enhancements. Our implementation of PipeSort includes the usual optimizations of aggregating and removing duplicates while sorting, instead of doing aggregation as a different phase after sorting[Gra93]. Often we can reduce the sorting cost by taking advantage of the partial sorting order. For instance, in Figure 3 for sorting ACD in the attribute order AD, we can get a sorted run of D for each distinct value of AC and for each distinct A we can merge these runs of D. Also, after the PipeSort algorithm has fixed the order in which each group-by is generated we can modify the sort-edges in the output search lattice to take advantage of the partial sorting orders whenever it is advantageous to do so.

3. Hash-based methods

We now discuss how we extend the hash-based method for computing a data cube. For hash-based methods, the new challenge is careful memory allocations of multiple hash-tables for incorporating optimizations **cache-results** and **amortize-scans**. For instance, if the hash tables for AB and AC fit in memory then the two group-bys could be computed in one scan of ABC. After AB is computed one could compute A and B while AB is still in memory and thus avoid the disk scan of AB. If memory were not a limitation, we could include all optimizations stated in Section 1 as follows.

Compute the bottom-most (level N) group-by in the lattice from raw data;
For $k = N - 1$ to 0
    For each $k + 1$ attribute group-by, g
        Compute in one scan of g all k attribute group-by for which g is the smallest parent;
        Save g to disk and release memory occupied by the hash table of g;

However, the data to be aggregated is usually too large for the hash-tables to fit in memory. The standard way to deal with limited memory when constructing hash tables is to partition the data on one or more attributes. When data is partitioned on some attribute, say A, then all group-bys that contain A can be computed by independently grouping on each partition — the results across multiple partitions need not be combined. We can share the cost of data partitioning across all group-bys that contain the partitioning attribute, leading to the optimization **share-partitions**. We present below the *PipeHash* algorithm that incorporates this optimization and also includes the optimizations **cache-results**, **amortize-scans** and **smallest-parent**.

Algorithm PipeHash. The input to the algorithm is the search lattice described in the previous section. The PipeHash algorithm first chooses for each group-by, the parent group-by with the smallest estimated total size. The outcome of this step is a minimum spanning tree (MST) where each vertex is a group-by and an edge from group-by a to b indicates that a is the smallest parent of b. In Figure 4 we show the MST for a four attribute search lattice (the size of each group-by is indicated below the group-by).

In general, the available memory will not be sufficient to compute all the group-bys in the MST together, hence the next step is to decide what group-bys to compute together, when to allocate and deallocate memory for different hash-tables, and what attribute to choose for partitioning data. We conjecture this problem to be NP-complete because solving this problem optimally requires us to solve the following sub-problem optimally: Divide the MST into smaller subtrees each of which can be computed in one scan of the group-by at the root of the MST such that the cost of scanning (from disk) the root group-by is minimized. This problem is similar to well-known NP-complete partitioning problems [GJ]. Hence, we resort to using a heuristic solution. Later (in Section 4) we show that our solution is very close to empirically estimated lower bounds for several datasets.

Optimizations cache-results and amortize-scans are favored by choosing as large a subtree of the MST as possible so that we can use the method above to compute together the group-bys in the subtree. However, when data needs to be partitioned based on some attribute, the partitioning attribute limits the subtree to only include group-bys containing the partitioning attribute. We therefore, choose a partitioning attribute that allows the choice of the largest subtree as shown in the pseudo-code of the PipeHash algorithm below.

PipeHash:

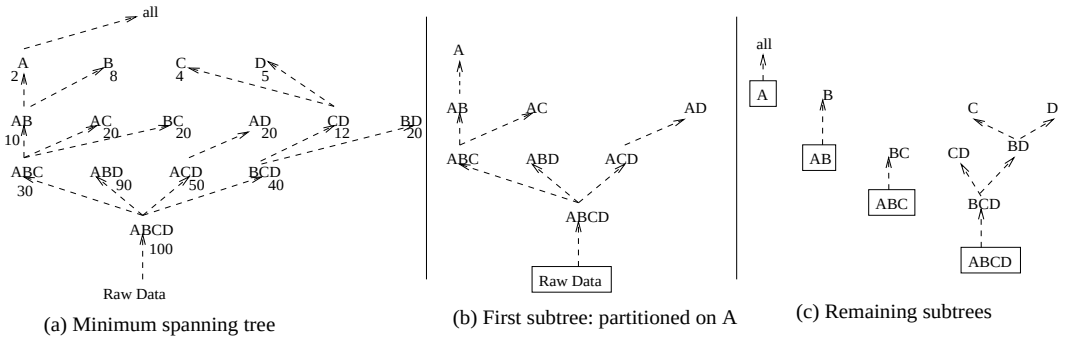


Figure 4: PipeHash on a four attribute group-by

(Input: search lattice with estimated size of each group-by)
Initialize worklist with MST of the search lattice;
While worklist is not empty
    Pick any tree T from the worklist;
    T' = Select-subtree of T to be executed next;
    Compute-subtree T';

Select-subtree
If memory required by $T <$ memory available, return T
Else, let S be the set of attributes of root(T)
    (We will pick $s \subset S$ for partitioning root of T. For any choice of s we get a subtree T_s of
        T also rooted at T and consisting of all group-bys that contain s.)
    Let P_s = maximum number of partitions of root(T) possible if partitioned on $s \subset S$;
    We choose $s \subset S$ such that
        memory required by $T_s/P_s <$ memory available, and
        T_s is the largest over all subsets of S;
    Remove T_s from T;
    This leaves $T - T_s$ which is a forest of smaller trees, add this to the worklist;
return T_s;

Compute-subtree
    numParts = (memory required by T')*(fudge_factor)/memory available;
    Partition root of T' into numParts;
    For each partition of root(T')
        For each node in T' (scanned in a breadth first manner)
            Compute all children of the node in one scan;
            If the node is cached, save it to disk and release memory occupied by its hash-table;

Example: Figure 4 illustrates the PipeHash algorithm for the four attribute search lattice of Figure 1. The boxed group-bys represent the root of the subtrees. Figure 4(a) shows the minimum spanning tree. Assume there is not enough memory to compute the whole tree in one pass and we need to partition the data. Figure 4(b) shows the first subtree T_A selected when A is chosen as the partitioning attribute. After removing T_A from the MST, we are left with four subtrees as shown in Figure 4(c). None of the group-bys in these subtrees include A. For computing T_A, we first partition the raw data

on A. For each partition we compute first the group-by $ABCD$; then scan $ABCD$ (while it is still in memory) to compute ABC, ABD and ACD together; save $ABCD$ and ABD to disk; compute AD from ACD; save ACD and AD to disk; scan ABC to compute AB and AC; save ABC and AC to disk; scan AB to compute A and save AB and A to disk. After T_A is computed, we compute each of the remaining four subtrees in the worklist.

Note that PipeHash incorporates the optimization share-partitions by computing from the same partition all group-bys that contain the partitioning attribute. Also, when computing a subtree we maintain all hash-tables of group-bys in the subtree (except the root) in memory until all its children are created. Also, for each group-by we compute its children in one scan of the group-by. Thus PipeHash also incorporate the optimizations amortize-scans and cache-results.[5]

PipeHash is biased towards optimizing for the smallest-parent. For each group-by, we first fix the smallest parent and then incorporate the other optimizations. For instance, in Figure 4(c), we could have computed BC from BCD instead of its smallest parent ABC and thus saved the extra scan on ABC. However, in practice, saving on sequential disk scans is less important than reducing the CPU cost of aggregation by choosing the smallest parent.

Hashing Structures. There are two classes of hash tables depending on whether or not we allow collisions within a bucket. The non-collision based hashing scheme reduces to a multidimensional array where the data attributes are hashed into contiguous unique integers, typically during a pre-pass of the data. For the collision based schemes, the hash function we use is concatenation of the first n_i bits for each attribute i of the tuple. We will use the term Hash-Grid to refer to the collision based scheme and the term Hash-Array to refer to the non-collision based scheme.

The Hash-Grid is expected to be better when the data distribution in the N-dimensional space of the attributes is sparse because the Hash-Array scheme will result in lots of empty cells. On the other hand, the Hash-Array scheme has smaller space overhead per hashed tuple. For the Hash-Array we need to store only the aggregate values for each tuple since the attributes of the tuple can be uniquely determined by the indices of the hashed cell. In contrast, for the Hash-Grid the entire tuple needs to be stored. We use the following method for choosing between the Hash-Array and Hash-Grid for a given group-by.

Consider a k attribute group-by $A_1A_2 \ldots A_k$. Let D_i be the number of distinct values along attribute i and M be the estimated number of tuples in the group-by. Further, let sizeof(aggregate column) denote the total space required for the aggregating columns and all intermediate results of the aggregate functions and sizeof(attribute) denote the size of each of grouping attribute. The memory required for the Hash-Array is

$$M_a = \prod_{i=1}^{k} D_i \times \text{sizeof(array cell)} + M \times \text{sizeof(aggregate column)}$$

[5] When we run out of memory because of data skew we need to release memory by freeing-up some hash-tables of cached group-bys. Note that we do not do any caching across partitions and across subtrees. Within a subtree we choose the hash-tables to be released in the following order:

i) completed hash-tables that will not be used for generating other group-bys

ii) completed hash-tables that will be used or are being used

iii) incomplete hash-tables if there are more than one being generated

iv) part of a hash-table if it is the only one being generated.

When incomplete hash-tables or part of a hash-table is freed, the parent group-by will need to be re-scanned from disk to generate the remaining group-bys or part of a group-by as the case may be.

whereas memory required for the Hash-Grid (assuming a perfect hash function) is

$$M_h = M \times \text{sizeof(hash bucket)} \times \text{Fugdefactor} + M \times (k \times \text{sizeof(attribute)} + \text{sizeof(aggregate column)}).$$

Based on these estimates we choose the method with the smaller memory requirement. When memory is not a constraint, the Hash-Array method is preferred since we do not need to compare values after hashing.

4. Experimental evaluation

In this section, we present the performance of our cube algorithms on several real-life datasets and analyze the behavior of these algorithms on tunable synthetic datasets. These experiments were performed on a RS/6000 250 workstation running AIX 3.2.5. The workstation had a total physical memory of 256 MB. We used a buffer of size 32 MB. The datasets were stored as flat files on a local 2GB SCSI 3.5" drive with sequential throughput of about 1.5 MB/second.

Datasets. Table 1 lists the five real-life datasets used in the experiments. These datasets were derived from sales transactions of various department stores and mail order companies. A brief description is given next. The datasets differ in the number of transactions, the number of attributes, and the number of distinct values for each attribute. For each attribute, the number within brackets denotes the number of its distinct values.

- **Dataset-A:** This data is about supermarket purchases. Each transaction has three attributes: store id(73), date(16) and item identifier(48510). In addition, two attributes cost and amount are used as aggregation columns. There are a total of 5.5 million transactions.
- **Dataset-B:** This data is from a mail order company. A sales transaction here consists of four attributes: the customer identifier(213972), the order date(2589), the product identifier(15836), and the catalog used for ordering(214). There are a total of 7.5 million transactions.
- **Dataset-C:** This is data about grocery purchases of customers from a supermarket. Each transaction has five attributes: the date of purchase(1092), the shopper type(195), the store code(415), the state in which the store is located(46) and the product group of the item purchased(118). There are a total of 9 million transactions.
- **Dataset-D:** This is data from a department store. Each transaction has five attributes: the store identifier(17), the date of purchase(15), the UPC of the product(85161), the department number(44) and the SKU number(63895). There are a total of 3 million transactions.
- **Dataset-E:** This data is also from a department store. Each transaction has total of six attributes: the store number(4), the date of purchase(15), the item number(26412), the business center(6), the merchandising group(22496) and a sequence number(255). A seventh attribute: the quantity of purchase was used as the aggregating column. The total number of transactions equaled 0.7 million.

Algorithms compared. For providing a basis of evaluation, we choose the straightforward method of computing each group-by in a cube as a separate group-by resulting in algorithms *NaiveHash* and *NaiveSort* depending on whether group-bys are computed using hash-based or sort-based methods. We further compare our algorithms against easy but possibly unachievable lower-bounds.

For the hash-based method the lower bound is obtained by summing up the following operations: Compute the bottom-most (level-N) group-by by hashing raw-data stored on disk; include the data

Dataset	Number of grouping attributes	number of tuples (in millions)	size (in MB)
Dataset-A	3	5.5	110
Dataset-B	4	7.5	121
Dataset-C	5	9	180
Dataset-D	5	3	121
Dataset-E	6	0.7	18

Table 1: Description of the datasets

partitioning cost if any. Compute all other group-bys by hashing the smallest parent assumed to be in memory; ignore data partitioning costs. Save all computed group-bys to disk.

For the sort-based method the lower bound is obtained by summing up the following operations: Compute the bottom-most (level-N) group-by by sorting the raw-data stored on disk. Compute all other group-bys from the smallest parent assumed to be in memory and sorted in the order of the group-by to be computed. Save all computed group-bys.

For both lower-bounds, we assume a perfect knowledge of the exact number of tuples in each group-by.

Performance results. Figure 5 shows the performance of the proposed PipeHash and PipeSort relative to the corresponding naive algorithms and estimated lower bounds. The total execution time is normalized by the time taken by the NaiveHash algorithm for each dataset to enable presentation on the same scale. For all datasets except Dataset-A we used Hash-Grid for the hash-based algorithms. We can make the following observations.

- Our algorithms are two to eight times faster than the naive methods.
- The performance of PipeHash is very close to our calculated lower bound for hash-based algorithms. The maximum difference in performance is 8%.
- The maximum gap between the performance of PipeSort and the calculated lower bound for the sort-based method is 22%.
- For most of the datasets, PipeHash is inferior to the PipeSort algorithms. We suspected this to be an artifact of these datasets. To further investigate the difference between them, therefore, we did a series of experiments on a synthetically generated dataset described next.

4.1. Comparing PipeSort and PipeHash

For the datasets in Table 1, the sort-based method performs better than the hash-based method. For Dataset-D, PipeSort is almost a factor of two better than PipeHash. Based on results in [GLS94], we had expected the hash-based method to be comparable or better than the sort-based method. Careful scrutiny of the performance data revealed that this deviation is because after some parent group-by is sorted we compute more than one group-by from it whereas for the hash-based method we build a different hash table for each group-by. Even though we share the partitioning cost for the hash-based method, the partitioning cost is not a dominant fraction of the total cost unlike sorting.

We conjectured that the hash-based method can perform better than the sort-based method when each group-by results in a considerable reduction in the number of tuples. This is because the cost of

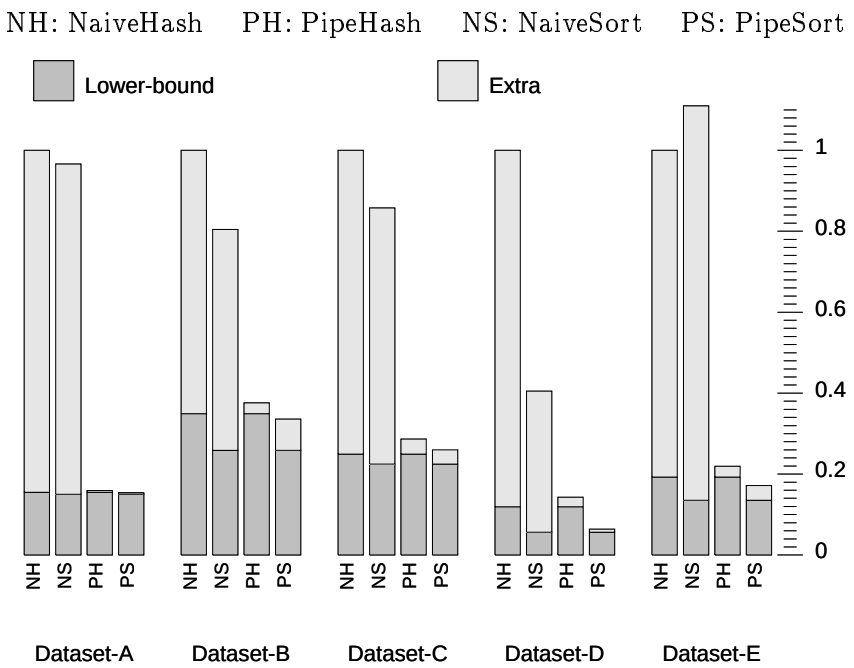


Figure 5: Performance of the cube computation algorithms on the five real life datasets. The y-axis denotes the total time normalized by the time taken by the NaiveHash algorithm for each dataset.

hashing at higher levels of aggregations can become a negligible fraction of the total cost when the number of tuples reduces rapidly. To validate our conjecture that the performance difference between the hash-based method and sort-based method is mainly due to the rate of decrease in the number of tuples as we aggregate along more and more attributes, we took a series of measurements on synthetic datasets described below.

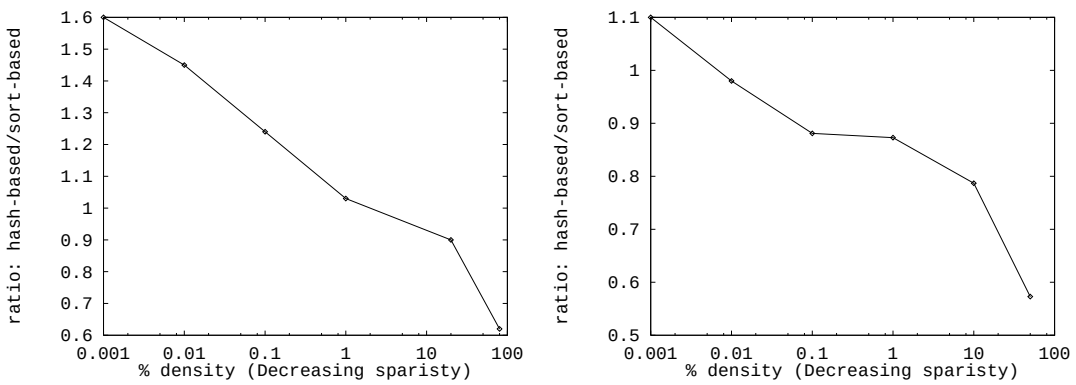
Synthetic datasets. Each dataset is characterized by four parameters:

1. Number of tuples, T.
2. Number of grouping attributes, N.
3. Ratio amongst the number of distinct values of each attribute $d_1 : d_2 : \ldots : d_N$.
4. A parameter, p, that provides a knob for altering the degree of sparsity of the data. It is defined as the ratio of T to the total number of possible attribute value combinations. Thus, if D_i denotes the number of distinct values of attribute i, then p is defined as $T/(D_1 \times D_2 \ldots D_N)$. Clearly, higher the degree of sparsity (lower value of p), lower the reduction in the number of tuples after aggregation.

Given these four parameters, the dataset is generated as follows. We first determine the total number of values D_i along each dimension i as:

$$D_i = \left(\frac{T}{p}\right)^{\frac{1}{N}} \frac{d_i}{(d_1 \times d_2 \times \ldots \times d_N)^{\frac{1}{N}}}$$

Then, for each of the T tuples, we choose a value for each of the N attributes from a uniform random distribution between 1 and the number of values for the attribute (i.e, D_i for attribute i).



(a) Ratio 1:2:4:20:300 (b) Ratio 1:1:1:1:1

Figure 6: Effect of sparseness on relative performance of the hash and sort-based algorithms for a 5 attribute synthetic dataset.

Results. We show the results for two sets of synthetic datasets. For each dataset in Figure 6(a) the total number of tuples is 5 million, the number of dimensions is 5, and the ratio between the number of distinct values of each attribute is 1:2:4:20:300 (large variance in number of distinct values). We vary the sparsity by changing p. The X-axis denotes decreasing levels of sparsity and the Y-axis denotes the ratio between the total running time of algorithms PipeHash and PipeSort. We notice that as the data becomes less and less sparse the hash-based method performs better than the sort-based method.

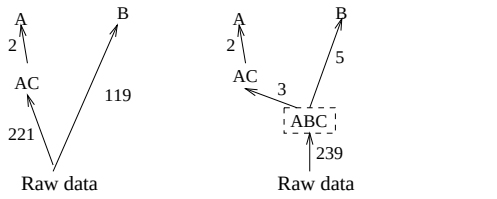
We repeated the same set of measurements for datasets with same number of tuples and dimensions as in Figure 6(a) but with a different ratio, 1:1:1:1:1 (Figure 6(b)). Comparing Figures 6(a) and (b), we notice the same trend for datasets with very different characteristics, empirically confirming that sparsity indeed is a predictor of the relative performance of the PipeHash and PipeSort algorithms.

5. Extensions

5.1. Partial cube

Very often the user is interested in only some of the group-by combinations instead of all possible combinations of attributes as in the data cube.

We can use the ideas developed for the cube algorithms to improve the performance of these computations. A trivial extension is to apply the PipeSort and PipeHash algorithm on the search lattice consisting of only group-bys that need to computed. However, more efficient computations can be achieved by generating intermediate group-bys even if they are not in the original requested subset. For instance, consider a four attribute cube. Even if a user is interested in only three group-bys ABC, ACD, BCD, it might be faster to first compute group-by $ABCD$ and then compute the three group-bys from $ABCD$ rather than compute each one of them from the raw data. We present extensions to the hash and sort-based methods that will consider such intermediate group-bys for computing a specified collection of group-bys.



(a) Best plan without extra nodes (b) Best Plan with additional node ABC

Figure 7: Computing group-by AC, A and B for the Synthetic dataset using the hash-based method.

Hash-based method. The input to the algorithm is (i) the search lattice G, modified to have edges between all group-by pairs (e, f) where f can be generated from e and not just between group-bys in adjacent levels, (ii) cost w on each edge (e, f) that denotes the cost of generating group-by f from e and (iii) a subset Y of the group-bys in G that needs to be computed. Our task is to find the minimal cost tree of G that includes at least all of Y and the starting raw data.

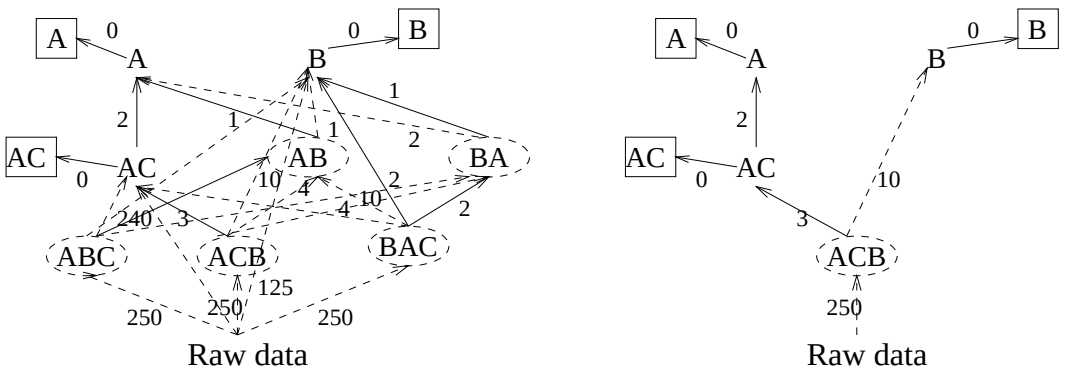
This problem is akin to the *minimal steiner tree* problem [GJ]. It is shown in [GJ] that the general steiner tree problem is NP-complete and [LN90] shows that it is NP-complete even for the special case of a cube graph that represents our search lattice. Hence, we will resort to approximate methods of computing the minimal cost tree. There are a number of existing approximate methods for the steiner tree problem and we borrow the one in [MSL80].

After we have found the minimal cost tree of G, we can use the PipeHash algorithm to compute the required group-bys. An illustration of the above scheme is given in Figure 7 for the Synthetic data set presented in Section 4.1(Figure 6(a), p=20%). Assume we wanted to compute group-bys A, B and AC. The best plan with the straight forward extension of the hashing method would take a total of 342 seconds whereas with the steiner tree method we would add the extra node ABC which results in a total cost of 249 seconds.

Sort-based method. For the sort-based method we cannot directly adapt the solution of the hash-based method since the cost of generating one group-by from another group-by g depends on the order in which g is sorted. Hence, we construct a different search lattice where for each group-by g of level k, $k!$ *sort-nodes* are added corresponding to each possible permutation of the attributes of g. The cost on an edge e from sort-node g to h denotes the cost of generating group-by sorted in order h from group-by sorted in order g. In addition, for each group-by g in set Y, we add a *select-node* and each of the sort-nodes of g are joined by a zero-cost edge to this select-node. We then apply the steiner tree algorithm to find the smallest tree including all the select-nodes and the raw data node.

The number of nodes in the search lattice using this method can be prohibitively large. However, a large number of nodes can easily be pruned. One pruning heuristic is to start from group-bys at level zero and start pruning sort-nodes level by level. For each level k, we only include sort-nodes that (i) have some prefix at higher levels and (ii) contain only attributes from sort-nodes of higher levels, except when the sort-node is of a group-by from set Y.

We illustrate this pruning step for the example of Figure 7. Figure 8(a) shows the pruned search lattice for the set Y consisting of A, B and AC. The select-nodes are within boxes and the intermediate sort-nodes are within ellipses. The numbers on edges indicates the time taken to compute one group-by from the other. At level 2, we have three sort-nodes AC, AB and BA. We have pruned CA, CD, CB, DE and all other nodes which have neither A nor B as prefix and hence violate condition (i). We have pruned BC, AD, BD and all other nodes containing attributes C, D or E since they violate



(a) Pruned search lattice

(b) Best plan after Steiner approximation

Figure 8: Computing group-by AC, A and B for the Synthetic dataset using the sort-based method.

condition (ii) except AC which belongs to set Y. Similarly, at level 3, we prune all sort-nodes except ABC, ACB and BAC. The best plan after applying the steiner tree algorithm on this search lattice is shown in Figure 8(b).

5.2. Hierarchies

It is common in OLAP applications for attributes to have hierarchies associated with them. For instance, going back to the example in Section 1, there is hierarchy `date` $\rightarrow$ `month` $\rightarrow$ `year` on attribute `date` and hierarchy `product` $\rightarrow$ `type` $\rightarrow$ `category` on attribute `product`. The requirement, in this case, is to compute group-bys for each combination of attributes along each level of the hierarchy.

The cube algorithms can be extended to handle attributes with hierarchies defined on them. We first construct a search lattice as follows. Start from the base level that contains just one group-by where each attribute is at the lowest level of detail of the hierarchy. From each group-by g, draw arcs to all other group-bys where exactly one of the attributes of g is at the next higher level of hierarchy. In Figure 9 we show the search lattice for two attributes A and B with hierarchy $A^1 \rightarrow A^2 \rightarrow A^3 \rightarrow$ `all` on A and the hierarchy $B^1 \rightarrow B^2 \rightarrow$ `all` on B.

Algorithms PipeHash and PipeSort can be applied to this search lattice with only a few modifications. The PipeHash algorithms changes only in the way partitions are handled. When partitioning on attribute A^i, we group-by on all attributes that contains any attribute below A^i in the hierarchy i.e., $A^1 \ldots A^i$. For the PipeSort algorithm, the sorting routine needs to be modified so that when data is sorted on attribute A^i it is sorted for all higher levels of the hierarchy, i.e., $A^i \ldots$ `all`.

6. Conclusion

We presented fast algorithms for computing multiple group-bys. We first concentrated on the computation of the data cube that requires all group-bys corresponding to each of the 2^N combinations of N attributes. We presented five optimizations *smallest-parent*, *cache-results*, *amortize-scans*, *share-sorts* and *share-partitions*, and embellished the sort-based and hash-based methods for computing single group-bys with these optimizations. These optimizations are often conflicting. Our proposed algorithms combine them so as to reduce the total cost. The sort-based algorithm, called *PipeSort*, develops the best plan by reducing the problem to a minimum weight matching problem on a bipartite graph. The hash-based algorithm, called *PipeHash*, develops the best plan by first creating the

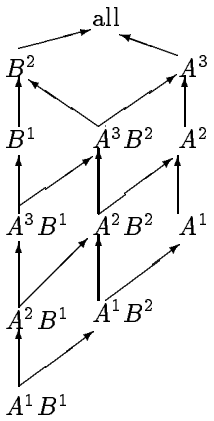


Figure 9: Attributes with hierarchy.

minimum spanning tree showing what group-by should be generated from what and then choosing a partitioning that takes into account memory availability.

Measurements on five real-life OLAP datasets yielded a factor of two to eight improvement with our algorithms over straightforward methods of computing each group-by separately. Although the PipeHash and PipeSort algorithms are not provably optimum, comparison with conservatively calculated lower bounds show that the PipeHash algorithm was within 8% and the PipeSort algorithm was within 22% of these lower bounds on several datasets. We further experimented with the PipeHash and PipeSort algorithms using a tunable synthetic dataset and observed that their relative performance depends on the sparsity of data values. PipeHash does better on low sparsity data whereas PipeSort does better on high sparsity data. Thus, we can choose between the PipeHash and PipeSort algorithms for a particular dataset based on estimated sparsity of the dataset.

We extended the cube algorithms to compute a specified subset of the 2^N group-bys instead of all of them. Our proposed extension considers intermediate group-bys that are not in the desired subset for generating the best plan. Our method reduces this optimization problem to the minimum steiner tree problem so that we can find globally good solutions for generating a specified list of group-bys. We also extended our algorithms for computing aggregations in the presence of hierarchies on attributes.

We expect the optimizations and algorithms developed in this paper to be very useful for OLAP databases because computation of multiple aggregates is a frequent operation not only during the data loading phase but also during ad-hoc query processing where fast response is crucial.

References.

[CCS93] E. F. Codd, S. B. Codd, and C. T. Salley. Beyond decision support. *Computerworld*, 27(30), July 1993.

[CM89] M.C. Chen and L.P. McNamee. The data model and access method of summary data management. *IEEE Transactions on Knowledge and Data Engineering*, 1(4):519–29, 1989.

[DANR96] P.M. Deshpande, S. Agarwal, J.F. Naughton, and R. Ramakrishnan. Computation of multidimensional aggregates. Technical Report Computer Science Technical Report TR1314, University of Wisconsin, Madison, Wisconsin, 1996.

[GBLP95] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tabs and sub-totals. Technical Report MSR-TR-95-22, Microsoft Research, Advance Technology Division, Microsoft Corporation, Redmond, Washington, November 1995.

[GGL95] R. L. Graham, M. Grötschel, and L. Lovasz, editors. *Handbook of combinatorics : volume 1*, chapter 8. Elsevier, Amsterdam, 1995.

[GHRU96] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Index selection for olap, 1996. Working Paper.

[GJ] M.R. Garey and D.S. Johnson. *Computers and Intractability*, chapter Appendix, pages 208–209.

[GLS94] Goetz Graefe, Ann Linville, and Leonard D. Shapiro. Sort versus hash revisited. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):934–944, 1994.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, Jun 1993.

[HNSS95] P.J. Haas, J.F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proceedings of the Eighth International Conference on Very Large Databases (VLDB)*, pages 311–22, Zurich, Switzerland, September 1995.

[HRU96] V. Harinarayan, A. Rajaraman, and J.D. Ullman. Implementing data cubes efficiently. In *Proc. of the ACM SIGMOD Conference on Management of Data*, June 1996.

[JS96] T. Johnson and D. Shasha. Hierarchically split cube forests for decision support: description and tuned design, 1996. Working Paper.

[LN90] X. Lin and L.M. Ni. Multicast communication in multicomputer networks. In *Proc. International Conference on Parallel Processing*, pages III–114–18, 1990.

[Mic92] Z. Michalewicz. *Statistical and Scientific Databases*. Ellis Horwood, 1992.

[MSL80] J. MacGregor Smith and J.S Liebman. An $o(n^2)$ heuristic algorithm for the directed steiner minimal tree problem. *Applied Mathematical Modelling*, 4(5):369–75, Oct 1980.

[PS82] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*, chapter 11, pages 247–254. 1982.

[Sho82] A. Shoshani. Statistical databases: Characteristics, problems and some solutions. In *Proceedings of the Eighth International Conference on Very Large Databases (VLDB)*, pages 208–213, Mexico City, Mexico, September 1982.

[SR96] B. Salzberg and A. Reuter. Indexing for aggregation, 1996. Working Paper.

[STL89] J. Srivastava, J.S.E. Tan, and V.Y. Lum. Tbsam: An access method for efficient processing of statistical queries. *IEEE Transactions on Knowledge and Data Engineering*, 1(4), 1989.

A. Procedure for estimating the size of group-bys

We give in this appendix the method we used for estimating the size of each group-by in our experiments. Our approach was to start with some rough estimates of the size of each group-by and refine these estimates during the course of computing the cube.

As suggested in [GBLP95], we first convert the raw data by mapping each attribute value into unique integers before starting to computing the cube. We can use this data conversion step, to get the number of distinct values, D_i for each attribute, A_i of the cube. A starting estimate of the size of group-by $A_1A_2 \ldots A_k$ consisting of attributes 1 through k can then be taken to be $\min(D_1 \times D_2 \times \ldots D_k, D)$.

These initial estimates are then refined in the following ways. (1) Once a group-by is computed, then the estimated size of all group-bys derivable from it can be made to be strictly smaller than this group-by. (2) Once we have covered two levels we can get better estimates as follows:

$$\frac{|ABCD|}{|ABC|} <= \frac{|ABD|}{|AB|} \quad Hence,\ |AB| <= \frac{|ABD||ABC|}{|ABCD|}$$

PipeHash and PipeSort algorithms can use these revised estimates to change their decisions about what group-by should be generated from what on the fly. We implemented only a limited amount of plan revising. The PipeSort algorithm before starting a pipeline re-computes the cost of generating it from the parent group-bys of the previous layer but does not change the selection of group-bys within a pipeline. Similarly, the PipeHash algorithm re-evaluates the parent group-by for generating the next sub-tree but does not otherwise change the structure of the sub-tree. We found that the gap in the performance of PipeSort and PipeHash based on estimates obtained using the above method was within 10% of when the algorithms were provided with perfect estimates of the size of each group-by.